CHAPTER

1 The History and Evolution of Java

To fully understand Java, one must understand the reasons behind its creation, the forces that shaped it, and the legacy that it inherits. Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission. While the remaining chapters of this book describe the practical aspects of Java—including its syntax, key libraries, and applications—this chapter explains how and why Java came about, what makes it so important, and how it has evolved over the years.

Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language. Computer language innovation and development occur for two fundamental reasons:

- To adapt to changing environments and uses
- To implement refinements and improvements in the art of programming

As you will see, the development of Java was driven by both elements in nearly equal measure.

Java's Lineage

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades. For these reasons, this section reviews the sequence of events and forces that led to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

The Birth of Modern Programming: C

The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you may know, when a computer language is designed, trade-offs are often made, such as the following:

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn't very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce "spaghetti code"—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs. (Specifically, given the standard dialects of Pascal available at the time, it was not practical to consider using Pascal for systems-level code.)

So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers' ability to produce it. A great deal of effort was being expended in academic circles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. Computer hardware was finally becoming common enough that a critical mass was being reached. No longer were computers kept behind locked doors. For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C's creation, the stage was set for a quantum leap forward in computer languages.

Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called BCPL, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s.

For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

The creation of C is considered by many to have marked the beginning of the modern age of computer languages. It successfully synthesized the conflicting attributes that had so troubled earlier languages. The result was a powerful, efficient, structured language that was relatively easy to learn. It also included one other, nearly intangible aspect: it was a *programmer's* language. Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. It was designed, implemented, and developed by real, working programmers, reflecting the way that they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. The result was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it. As such, it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers. As you will see, Java inherited this legacy.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, at the time it was hardly a language that encouraged clear and easy-to-understand programs. The 1960s gave birth to *structured programming*. This is the method of programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called *object-oriented programming (OOP)*. Object-oriented programming is discussed in detail later in this book, but here is a brief definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

In the final analysis, although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once the size of a program exceeds a certain point, it becomes so complex that it is difficult to grasp as a totality. While the precise size at which this occurs differs, depending upon both the nature of the program and the programmer, there is always a threshold at which a program becomes unmanageable. C++ added features that enabled this threshold to be broken, allowing programmers to comprehend and manage larger programs.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language "C with Classes." However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C's features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

The Stage Is Set for Java

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages at the time) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent

language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (platform-independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems. Further, because (at that time) much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritating but low-priority problem had become a high-profile necessity.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a

C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

As mentioned at the start of this chapter, computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. In the final analysis, though, it was not the individual features of Java that made it so remarkable. Rather, it was the language as a whole. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

The C# Connection

The reach and power of Java continues to be felt throughout the world of computer language development. Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore.

Perhaps the most important example of Java's influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java. For example, both share the same general syntax, support distributed programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall "look and feel" of these languages is very similar. This "cross-pollination" from Java to C# is the strongest testimonial to date that Java redefined the way we think about and use a computer language.

How Java Impacted the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let's look more closely at each of these.

Java Applets

At the time of Java's creation, one of its most exciting features was the applet. An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed inside a Java-compatible web browser. If the user clicks a link that contains an applet, the applet will download and run in the browser. Applets were intended to be small programs. They were typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allowed some functionality to be moved from the server to the client.

The creation of the applet was important because, at the time, it expanded the universe of objects that could move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

In the early days of Java, applets were a crucial part of Java programming. They illustrated the power and benefits of Java, added an exciting dimension to web pages, and enabled programmers to explore the full extent of what was possible with Java. Although it is likely that there are still applets in use today, over time they became less important. For reasons that will be explained, beginning with JDK 9, the phase-out of applets began, with applet support being removed by JDK 11.

Security

As desirable as dynamic, networked programs are, they can also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each, beginning with security.

As you are likely aware, every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable programs to be safely downloaded and executed on the client computer, it was necessary to prevent them from launching such an attack.

Java achieved this protection by enabling you to confine an application to the Java execution environment and prevent it from accessing other parts of the computer. (You will see how this is accomplished shortly.) The ability to download programs with a degree of confidence that no harm will be done may have been the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. In other words, a mechanism that allows the same application to be downloaded and executed by a wide variety of CPUs, operating systems, and browsers is required. It is not practical to have different versions of the application for different computers.

The *same* application code must work on *all* computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Java's Magic: The Bytecode

The key that allowed Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by what is called the *Java Virtual Machine (JVM)*, which is part of the Java Runtime Environment (JRE). In essence, the original JVM was designed as an *interpreter for bytecode*. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. Here is why.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once a JRE exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it manages program execution. Thus, it is possible for the JVM to create a restricted execution environment, called the *sandbox*, that contains the program, preventing unrestricted access to the machine. Safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that an entire Java program is not compiled into executable code all at once. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

One other point: There has been experimentation with an *ahead-of-time* compiler for Java. Such a compiler can be used to compile bytecode into native code *prior* to execution by the JVM, rather than on-the-fly. Some previous versions of the JDK supplied an experimental ahead-of-time compiler; however, JDK 17 has removed it. Ahead-of-time compilation is a specialized feature, and it does not replace Java's traditional approach just described. Because of the highly specialized nature of ahead-of-time compilation, it is not discussed further in this book.

Moving Beyond Applets

At the time of this writing, it has been more than two decades since Java's original release. Over those years, many changes have taken place. At the time of Java's creation, the Internet was a new and exciting innovation; web browsers were undergoing rapid development and refinement; the modern form of the smart phone had not yet been invented; and the near ubiquitous use of computers was still a few years off. As you would expect, Java has also changed and so, too, has the way that Java is used. Perhaps nothing illustrates the ongoing evolution of Java better than the applet.

As explained previously, in the early years of Java, applets were a crucial part of Java programming. They not only added excitement to a web page, they were also a highly visible part of Java, which added to its charisma. However, applets rely on a Java browser plug-in. Thus, for an applet to work, the browser must support it. Over the past few years, support for the Java browser plug-in has been waning. Simply put, without browser support, applets are not viable. Because of this, beginning with JDK 9, the phase-out of applets was begun, with support for applets being deprecated. In the language of Java, *deprecated* means that a feature is still available but flagged as obsolete. Thus, a deprecated feature should not be used for new code. The phase-out became complete with the release of JDK 11 because run-time support for applets was removed. Beginning with JDK 17, the entire Applet API was deprecated for removal.

As a point of interest, a few years after Java's creation an alternative to applets was added to Java. Called Java Web Start, it enabled an application to be dynamically downloaded from a web page. It was a deployment mechanism that was especially useful for larger Java applications that were not appropriate for applets. The difference between an applet and a Web Start application is that a Web Start application runs on its own, not inside the browser. Thus, it looks much like a "normal" application. It does, however, require that a stand-alone JRE that supports Web Start is available on the host system. Beginning with JDK 11, Java Web Start support has been removed.

Given that neither applets nor Java Web Start are supported by modern versions of Java, you might wonder what mechanism should be used to deploy a Java application. At the time of this writing, part of the answer is to use the **jlink** tool added by JDK 9. It can create a complete run-time image that includes all necessary support for your program, including the JRE. Another part of the answer is the **jpackage** tool. Added by JDK 16, it can be used to create a ready-to-install application. Although a detailed discussion of deployment strategies is outside the scope of this book, it is something that you will want to pay close attention to going forward.

A Faster Release Schedule

Another major change has recently occurred in Java, but it does not involve changes to the language or the run-time environment. Rather, it relates to the way that Java releases are scheduled. In the past, major Java releases were typically separated by two or more years. However, subsequent to the release of JDK 9, the time between major Java releases has been decreased. Today, it is anticipated that a major release will occur on a strict time-based schedule, with the expected time between such releases being just six months.

Each six-month release, now called a *feature release,* will include those features ready at the time of the release. This increased *release cadence* enables new features and enhancements to be available to Java programmers in a timely fashion. Furthermore, it allows Java to respond quickly to the demands of an ever-changing programming environment. Simply put, the faster release schedule promises to be a very positive development for Java programmers.

At three-year intervals, it is anticipated that a *long-term support* (LTS) release will take place. An LTS release will be supported (and thus remain viable) for a period of time longer than six months. The first LTS release was JDK 11. The second LTS release was JDK 17, for which this book has been updated. Because of the stability that an LTS release offers, it is likely that its feature set will define a baseline of functionality for a number of years. Consult Oracle for the latest information concerning long-term support and the LTS release schedule.

Currently, feature releases are scheduled for March and September of each year. As a result, JDK 10 was released in March 2018, which was six months after the release of JDK 9. The next release (JDK 11) was in September 2018. JDK 11 was an LTS release. This was followed by JDK 12 In March 2019, JDK 13 in September 2019, and so on. At the time of this writing, the latest release is JDK 17, which is an LTS release. Again, it is anticipated that every six months a new feature release will take place. Of course, you will want to consult the latest release schedule information.

At the time of this writing, there are a number of new Java features on the horizon. Because of the faster release schedule, it is very likely that several of them will be added to Java over the next few years. You will want to review the information and release notes provided by each six-month release in detail. It is truly an exciting time to be a Java programmer!

Servlets: Java on the Server Side

Client side code is just one half of the client/server equation. Not long after the initial release of Java, it became obvious that Java would also be useful on the server side. One result was the *servlet.* A servlet is a small program that executes on the server.

Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content was available through mechanisms such as CGI (Common Gateway Interface), the servlet offered several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container. Today, server-side code in general constitutes a major use of Java.

The Java Buzzwords

No discussion of Java's history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java managed to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, were kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to

create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer will often manually allocate and free dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform

solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend( )**, **resume( )**, and **stop( )** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Annotations
- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style **for** loop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities

This is not a list of minor tweaks or incremental upgrades. Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced **for**, and varargs, introduced new syntax elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language. Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself.

The importance of these new features is reflected in the use of the version number "5." The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn't seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer's kit was called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the *developer version* number. The "5" in J2SE 5 is called the *product version* number.

The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java platform. First, notice that the "2" was dropped. Thus, the platform was now named *Java SE*, and the official product name was *Java Platform, Standard Edition 6*. The Java Development Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several new packages, and offered improvements to the run time. It also went through several

updates during its (in Java terms) long life cycle, with several upgrades added along the way. In general, Java SE 6 served to further solidify the advances made by J2SE 5.

Java SE 7 was the next release of Java, with the Java Development Kit being called JDK 7, and an internal version number of 1.7. Java SE 7 was the first major release of Java after Sun Microsystems was acquired by Oracle. Java SE 7 contained many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages were also included, but it is the language and library additions that were of most interest to Java programmers.

The new language features were developed as part of *Project Coin*. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these features were collectively referred to as "small," the effects of these changes have been quite large in terms of the code they impact. In fact, for many programmers, these changes may well have been the most important new features in Java SE 7. Here is a list of the language features added by JDK 7:

- A **String** can now control a **switch** statement.
- Binary integer literals.
- Underscores in numeric literals.
- An expanded **try** statement, called ***try****-with-resources*, that supports automatic resource management. (For example, streams can be closed automatically when they are no longer needed.)
- Type inference (via the *diamond* operator) when constructing a generic instance.
- Enhanced exception handling in which two or more exceptions can be caught by a single **catch** (multi-catch) and better type checking for exceptions that are rethrown.
- Although not a syntax change, the compiler warnings associated with some types of varargs methods were improved, and you have more control over the warnings.

As you can see, even though the Project Coin features were considered small changes to the language, their benefits were much larger than the qualifier "small" would suggest. In particular, the **try***-with-resources* statement has profoundly affected the way that stream-based code is written. Also, the ability to use a **String** to control a **switch** statement was a long-desired improvement that simplified coding in many situations.

Java SE 7 made several additions to the Java API library. Two of the most important were the enhancements to the NIO Framework and the addition of the Fork/Join Framework. NIO (which originally stood for *New I/O*) was added to Java in version 1.4. However, the changes added by Java SE 7 fundamentally expanded its capabilities. So significant were the changes, that the term *NIO.2* is often used.

The Fork/Join Framework provides important support for *parallel programming*. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems. The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addressed parallel programming by:

- Simplifying the creation and use of tasks that can execute concurrently
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scaleable applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

The next release of Java was Java SE 8, with the developer's kit being called JDK 8. It has an internal version number of 1.8. JDK 8 was a significant upgrade to the Java language because of the inclusion of a far-reaching new language feature: the *lambda expression*. The impact of lambda expressions was, and will continue to be, profound, changing both the way that programming solutions are conceptualized and how Java code is written. As explained in detail in Chapter 15, lambda expressions add functional programming features to Java. In the process, lambda expressions can simplify and reduce the amount of source code needed to create certain constructs, such as some types of anonymous classes. The addition of lambda expressions also caused a new operator (the –>) and a new syntax element to be added to the language.

The inclusion of lambda expressions has also had a wide-ranging effect on the Java libraries, with new features being added to take advantage of them. One of the most important was the new stream API, which is packaged in **java.util.stream**. The stream API supports pipeline operations on data and is optimized for lambda expressions. Another new package was **java.util.function**. It defines a number of *functional interfaces*, which provide additional support for lambda expressions. Other new lambda-related features are found throughout the API library.

Another lambda-inspired feature affects **interface**. Beginning with JDK 8, it is now possible to define a default implementation for a method specified by an interface. If no implementation for a default method is created, then the default defined by the interface is used. This feature enables interfaces to be gracefully evolved over time because a new method can be added to an interface without breaking existing code. It can also streamline the implementation of an interface when the defaults are appropriate. Other new features in JDK 8 include a new time and date API, type annotations, and the ability to use parallel processing when sorting an array, among others.

The next release of Java was Java SE 9. The developer's kit was called JDK 9. With the release of JDK 9, the internal version number is also 9. JDK 9 represented a major Java release, incorporating significant enhancements to both the Java language and its libraries. Like the JDK 5 and JDK 8 releases, JDK 9 affected the Java language and its API libraries in fundamental ways.

The primary new JDK 9 feature was *modules,* which enable you to specify the relationship and dependencies of the code that comprises an application. Modules also add another dimension to Java's access control features. The inclusion of modules caused a new syntax element and several keywords to be added to Java. Furthermore, a tool called **jlink** was added to the JDK, which enables a programmer to create a run-time image of an application that contains only the necessary modules. A new file type, called JMOD, was created. Modules also have a profound affect on the API library because, beginning with JDK 9, the library packages are now organized into modules.

Although modules constitute a major Java enhancement, they are conceptually simple and straightforward. Furthermore, because pre-module legacy code is fully supported, modules can be integrated into the development process on your timeline. There is no need

to immediately change any preexisting code to handle modules. In short, modules added substantial functionality without altering the essence of Java.

In addition to modules, JDK 9 included many other new features. One of particular interest is JShell, which is a tool that supports interactive program experimentation and learning. (An introduction to JShell is found in Appendix B.) Another interesting upgrade is support for private interface methods. Their inclusion further enhanced JDK 8's support for default methods in interfaces. JDK 9 added a search feature to the **javadoc** tool and a new tag called **@index** to support it. As with previous releases, JDK 9 contained a number of enhancements to Java's API libraries.

As a general rule, in any Java release, it is the new features that receive the most attention. However, there was one high-profile aspect of Java that was deprecated by JDK 9: applets. Beginning with JDK 9, applets were no longer recommended for new projects. As explained earlier in this chapter, because of waning browser support for applets (and other factors), JDK 9 deprecated the entire applet API.

The next release of Java was Java SE 10 (JDK 10). As explained earlier, beginning with JDK 10, Java releases are anticipated to occur on a strict time-based schedule, with the time between major releases expected to be just six months. As a result, JDK 10 was released in March 2018, which was six months after the release of JDK 9. The primary new language feature added by JDK 10 was support for *local variable type inference*. With local variable type inference, it is now possible to let the type of a local variable be inferred from the type of its initializer, rather than being explicitly specified. To support this new capability, the context-sensitive keyword **var** was added to Java. Type inference can streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer. It can also simplify declarations in cases in which the type is difficult to discern or cannot be explicitly specified. Local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up-to-date with evolving trends in language design. Along with a number of other changes, JDK 10 also redefined the Java version string, changing the meaning of the version numbers so that they better align with the new time-based release schedule.

The next version of Java was Java SE 11 (JDK 11). It was released in September 2018, which was six months after JDK 10. JDK 11 was an LTS release. The primary new language feature in JDK 11 was support for the use of **var** in a lambda expression. Along with a number of tweaks and updates to the API in general, JDK 11 added a new networking API, which will be of interest to a wide range of developers. Called the *HTTP Client API,* it is packaged in **java.net.http**, and it provides enhanced, updated, and improved networking support for HTTP clients. Also, another execution mode was added to the Java launcher that enables it to directly execute simple single-file programs. JDK 11 also removed some features. Perhaps of the greatest interest because of its historical significance is the removal of support for applets. Recall that applets were first deprecated by JDK 9. With the release of JDK 11, applet support has been removed. Support for another deployment-related technology called Java Web Start was also removed from JDK 11. As the execution environment has continued to evolve, both applets and Java Web Start were rapidly losing relevance. Another key change in JDK 11 is that JavaFX was no longer included in the JDK. Instead, this GUI framework has become a separate open-source project. Because these features are no longer part of the JDK, they are not discussed in this book.

Between the JDK 11 LTS and the next LTS release (JDK 17) were five feature releases: JDK 12 through JDK 16. JDK 12 and JDK 13 did not add any new language features. JDK 14 added support for the **switch** expression, which is a **switch** that produces a value. Other enhancements to **switch** were also included. Text blocks, which are essentially string literals that can span more than one line, were added by JDK 15. JDK 16 enhanced **instanceof** with pattern matching and added a new type of class called a *record* along with the new context-sensitive keyword **record**. A record provides a convenient means of aggregating data. JDK 16 also supplied a new application packaging tool called **jpackage**.

At the time of this writing, Java SE 17 (JDK 17) is the latest version of Java. As mentioned, it is the second LTS Java release. Thus, it is of particular importance. Its major new feature is the ability to seal classes and interfaces. Sealing gives you control over the inheritance of a class and the inheritance and implementation of an interface. To this end, it adds the context-sensitive keywords **sealed**, **permits**, and **non-sealed**, which is the first hyphenated Java keyword. JDK 17 marks the applet API as deprecated for removal. As explained, support of applets was removed several years ago. However, the applet API was simply deprecated, which allowed vestigial code that relied on this API to still compile. With the release of JDK 17, the applet API is now subject to removal by a future release.

One other point about the evolution of Java: Beginning in 2006, the process of open-sourcing Java began. Today, open-source implementations of the JDK are available. Open-sourcing further contributes to the dynamic nature of Java development. In the final analysis, Java's legacy of innovation is secure. Java remains the vibrant, nimble language that the programming world has come to expect.

The material in this book has been updated through JDK 17. Many new Java features, updates, and additions are described throughout. As the preceding discussion has highlighted, however, the history of Java programming is marked by dynamic change. You will want to review the new features in each subsequent Java release. Simply put: The evolution of Java continues!

A Culture of Innovation

Since the beginning, Java has been at the center of a culture of innovation. Its original release redefined programming for the Internet. The Java Virtual Machine (JVM) and bytecode changed the way we think about security and portability. Portable code made the Web come alive. The Java Community Process (JCP) redefined the way that new ideas are assimilated into the language. The world of Java has never stood still for very long. JDK 17 is the latest release in Java's ongoing, dynamic history.